

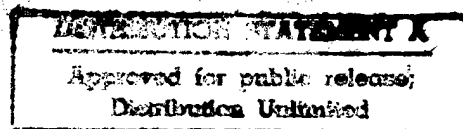


Carnegie Mellon
Software Engineering Institute

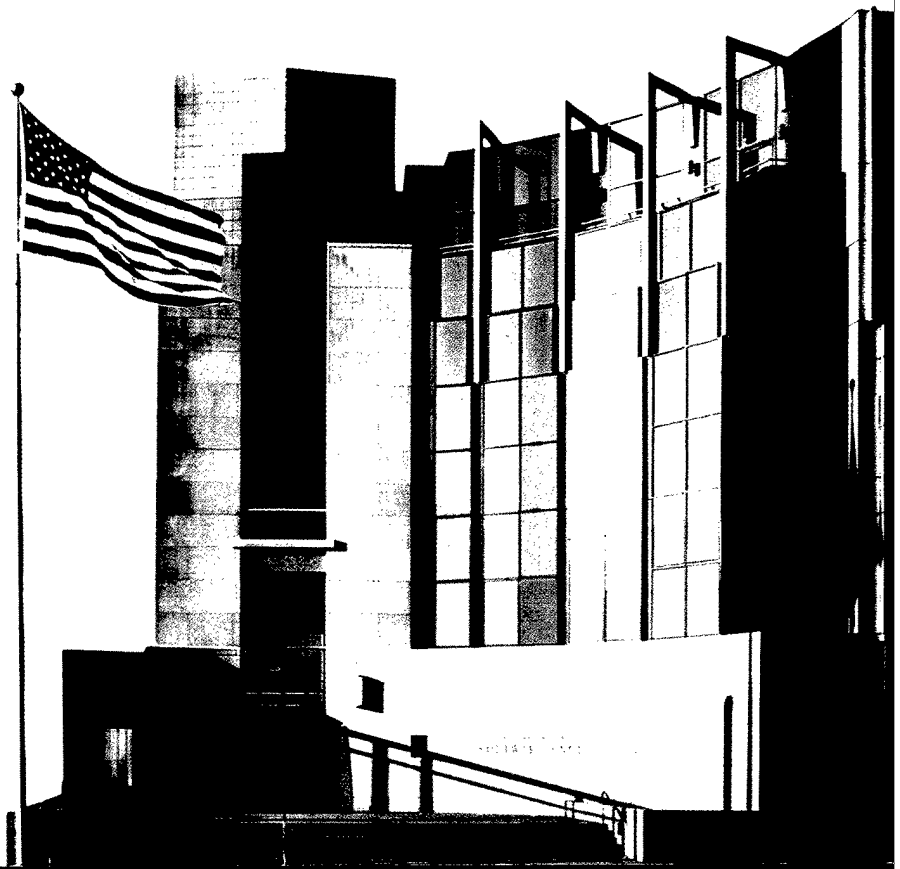
Mapping MetaH into ACME

Mario R. Barbacci
Charles B. Weinstock

July 1998



SPECIAL REPORT
CMU/SEI-98-SR-006



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.



Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

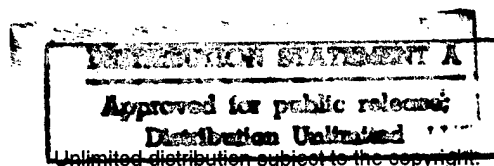
Mapping MetaH into ACME

CMU/SEI-98-SR-006

Mario R. Barbacci
Charles B. Weinstock

July 1998

Architecture Tradeoff Analysis Initiative



19980810 153

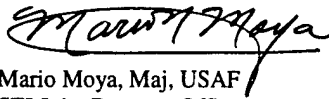
DTIC QUALITY INSPECTED 1

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Mario Moya, Maj, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright 1998 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: <http://www.asset.com> / e-mail: sei@asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800 225-3842.

Table of Contents

1	Introduction to MetaH and ACME	1
2	Software Architecture Specification	3
2.1	The ACME "MetaH_Family"	3
2.2	Interface and Implementation Classes	5
2.2.1	Type Packages	5
2.2.2	Subprograms	6
2.2.3	Packages	6
2.2.4	Monitors	6
2.2.5	Processes	6
2.2.6	Macros	8
2.2.7	Modes	8
2.2.8	Applications	10
2.2.9	Error Models	10
2.3	Component Declarations	11
2.3.1	Component Classes, Types, and Subclasses	11
2.3.1.1	Port Type and Subclass Declaration	11
2.3.1.2	Event Subclass Declaration	11
2.3.1.3	Process Subclass Declaration	13
2.3.1.4	Mode Subclass Declaration	14
2.3.1.5	State Subclass Declaration	14
2.3.2	Component Visibility and Naming	14
2.4	Connection Declarations	14
2.4.1	Port Connections	15
2.4.2	Event Connections	15
2.4.2.1	Event to Event Connections	16
2.4.2.2	Event to Process Connections	16
2.4.2.3	Event to Mode Connections	17
2.4.3	Equivalence Connections	17
2.4.4	Path Declarations	17
2.4.4.1	Execution Paths	17
2.4.4.2	Error Paths	18
2.4.5	Attribute Assignments	18
3	A Complete MetaH Example	19

4 The MetaH Example Translated into ACME	21
5 Generating Rapide Behavioral Specifications from MetaH Descriptions	25
References	27

List of Figures

Figure 2-1	MH_Family	4
Figure 2-2	MetaH Type Package and Translation into ACME	7
Figure 2-3	MetaH Process and Translation into ACME	9
Figure 2-4	MetaH Interface and Implementation with Components and Translation into ACME	12
Figure 2-5	MetaH Port Declaration and Translation into ACME	13
Figure 2-6	MetaH Port Connection and Translation into ACME	16

Abstract

This report explores the translation of MetaH into ACME as a first step into the translation of MetaH to other architecture description languages (e.g., Rapide) to take advantage of any toolsets developed for the target language. We start by comparing the meta-models of ACME and MetaH, we establish mapping rules for each MetaH construct, and we present a full MetaH example taken from the MetaH Library at Honeywell. The report concludes with a brief description of possible alternative paths to obtain (limited) Rapide behavioral specifications from MetaH timing and sequencing of operations.

1 Introduction to MetaH and ACME

The ACME interchange format was originally conceived as a way to share tool capabilities provided by a particular ADL with other ADLs, while avoiding the production of many pairwise language translators. ACME has been embraced the DARPA EDCS Program as a “domain-neutral” architecture description language for building a core set of architectural tools and as a common core representation for more domain-specific ADLs.

The basic aim of translating architectural design descriptions in an ADL to and from the ACME interchange format is to provide access to emerging tool capabilities without having to produce a redundant architectural specification in their native ADLs. In general, such translators present two basic difficulties:

- meta-model differences between an ADL and ACME
- attribute labeling and semantic differences between ADLs

ACME’s meta-model includes a first-class connector concept, whereas not all ADLs support such a concept. ADLs that do not support first-class connectors do, by necessity, support alternative interconnection or configuration concepts. The challenge of producing translators for these ADLs is to determine what kinds of simple or complex connectors can be used in ACME for whatever the ADL provides to describe component inter-relationships, while preserving their structural and coordination semantics. For the ADLs that have been used as exemplars in the development of ACME—UniCon and Rapide, for example—reasonable solutions to this problem have been found.

Attributes present more significant challenges to constructing translators. Non-structural design information, which is typically the subject of modeling, analysis, and generation tools, is expressed in ACME by attribute, or property lists. ACME is not concerned with this information per se, but merely associates it with elements of the structural description and communicates it between translations. The problems arise between two different ADLs using ACME as an interchange format. The two may use the same attribute label for different information, different attribute labels for the same information, or may provide the same information in different ways. They may also provide the same information in the same way, but have subtle (or not so subtle) differences in interpretation within the context of particular tools. As a facet of the ongoing development of ACME, although not directly related to its design, it has been proposed that a core set of attributes with consistent semantics across ADLs be adopted. In the interim, issues of agreement must be considered pairwise between ADLs, and in the longer term such issues will remain for attributes outside of the core set.

The MetaH architectural description language (ADL) and associated toolset support architectural modeling of embedded real-time system applications. Based on the architectural model the toolset performs syntactic consistency checks regarding the architectural structure, schedulability analysis of the real-time tasks, and generates executable code that integrates application components into a communicating set of processes, complemented with a runtime system that contains a tailored scheduler and communication support.

MetaH focuses on system integration and is designed to interface with more specialized toolsets that produce functional components (in the paper, "An aspect of MetaH extensibility is the ability to interface with other ADLs and associated analysis capabilities.") These are often called DSLs, but I think they often have the flavor of specialized ADLs. For example, one could think of integrating the USC/ISI message handling code generator with MetaH, so that MetaH could be used to integrate message-handling subsystems generated by those tools into an overall embedded system [Vestal 98].

Honeywell has used MetaH as an architectural language in a number of application settings and has extended and completed the modeling capability with specialized sub-languages and tools. An aspect of MetaH extensibility is the ability to interface with other ADLs and associated analysis capabilities.

This report explores the translation of MetaH into ACME as a first step into the translation of MetaH to other ADLs (e.g., Rapide) to take advantage of any toolsets developed for the target language. We start by comparing the meta-models of ACME and MetaH, we establish mapping rules for each MetaH construct, and we present a full MetaH example taken from the MetaH Library at Honeywell. The report concludes with a brief description of possible alternative paths to obtain (limited) Rapide behavioral specifications from MetaH timing and sequencing of operations.

We appreciate the comments and suggestions on earlier drafts of this report offered by Peter Feiler (CMU/SEI) and Steve Vestal (Honeywell). Dave Garland (CMU/SCS) and his team made available to us the AcmeStudio tools used to test the translations.

2 Software Architecture Specification

This chapter explains the syntax and semantics of the MetaH architecture description language and their translation into ACME. We have tried to the maximum extent to retain the structure of Chapter 5 in the *MetaH Programmer's Manual* [MetaH 98] to check the completeness of the approach, i.e., we have accounted for each and every MetaH feature.

A MetaH software architecture specification for an application consists of a series of interface and implementation specifications (the components), together with a single application specification that describes the connections between components. The interface and implementation specifications explain how types of objects are constructed from previously specified types of objects. The application specification combines a software object with a hardware object (the execution platform) and type packages (the data types exchanged between components) to specify a complete system architecture.

We retain this flavor in the translation to ACME. MetaH interface specifications are translated to ACME component types. MetaH implementation specifications are translated into ACME component types extending the previous type. MetaH objects are translated into ACME instances of the appropriate type (i.e., the type derived from the implementation specification). Finally, a MetaH application is also translated into an ACME component type. After the MetaH description is translated, a single ACME system is declared as an instance of the ACME component type generated from the MetaH application.

Whenever we discuss ACME and MetaH constructs with similar names (e.g., “Port”) we will prefix the name of the construct with the name of the language (e.g., “ACME Port”) to make explicit which flavor of the construct we are talking about.

2.1 The ACME “MetaH_Family”

To facilitate the translation to ACME we use the ACME Family construct to declare a collection of standard MetaH-related types, types used in any MetaH to ACME translation. This family (“MetaH_Family,” Figure 2-1) is augmented with the (translated) MetaH type packages, interfaces, implementations, and application.

```

Family MetaH_Family () =
{ /* BEGIN STANDARD METAH DECLARATIONS */

property type MH_mode_subclass =
    enum {MH_initial, MH_other};
property type MH_port_subclass =
    enum {MH_in, MH_out};
property type MH_process_subclass =
    enum {MH_periodic, MH_aperiodic};
property type MH_event_subclass =
    enum {MH_interrupt, MH_signal, MH_nudge, MH_node};
property type MH_execution_path = sequence;
property type MH_error_path = sequence
property type MH_Implementation_name = string;
property type MH_Interface_name = string;

port type MH_port = {};
port type MH_event = {};

component type MH_mode =
    {port MH_event_port: MH_port
      = {property MH_port_subclass = MH_in;}};
component type MH_macro = {};
component type MH_monitor = {};
component type MH_package = {};
component type MH_subprogram = {};
component type MH_process =
    {port MH_event_port: MH_port
      = {property MH_port_subclass = MH_in;}};
component type MH_error_model = {};
component type MH_error_state = {};

connector type MH_connector =
    {roles {MH_source; MH_sink};
    property MH_port_identifier: string;};
    /* The MetaH connector roles are wired-in */

/* BEGIN EXAMPLE SPECIFIC DECLARATIONS */
.....
}; /* End MetaH_Family */

System identifier: MetaH_Family =
    {component identifier = new identifier;};

```

Figure 2-1: MH_Family

The ACME system declared at the end of the translation uses the declarations in `MH_Family` and declares an instance of the ACME component type derived from the MetaH application. This should become clear through the examples provided in the report.

2.2 Interface and Implementation Classes

MetaH interface and implementation specifications can be divided into two main groups based on the class of object being specified: source objects (type packages, subprograms, packages, and monitors) and higher-level objects (processes, macros, modes, and applications) ([MetaH 98], Section 5.1.3). We make no distinction between these two groups and how they are translated into ACME.

With three exceptions, all MetaH constructs can be translated one-to-one into ACME constructs. That is, we can translate interfaces, implementations, and systems into ACME types as they occur. Although there is no requirement that the translation be done “on the fly”, it is reassuring that this simple rule applies!

The first two exceptions to the rule are MetaH type packages interfaces (Section 2.2.1) and MetaH interfaces (Section 2.3) with internal components. In both cases we must wait until we have the corresponding implementation before translating the construct into ACME. The third exception are MetaH equivalence connections (Section 2.4.3). We need to go back and annotate sets of components with an ACME attribute identifying them as members of a set.

2.2.1 Type Packages

A MetaH type package consists of a collection of type declarations for input and output buffer variables. These buffer variables are the ports. ([MetaH 98], Section 5.1.3.1.1)

Type packages¹ are one case in which a MetaH entity does not map directly into one ACME entity. We have to break the type package so that each individual type identifier is translated into an ACME port type declaration (i.e., a MetaH type package is a collection of data type declarations, each of which is translated into a different ACME port type.) We need to do this bundling and breaking because each data type must be annotated with its own individual MetaH attributes. To achieve the same effect in ACME we make the variables first class citizens, i.e., ACME port types and then we annotate them with ACME properties.

For example, consider the following MetaH type package in Figure 2-2 (a). The MetaH type package “`PORT_TYPES`” declares two data types (“`ANY_TYPE`” and “`ANOTHER_TYPE`”). Each of these is translated into a separate ACME port type, declared by extending the basic

1. Type packages were known as “port type modules” in previous version of the MetaH documentation and toolset.

type `MH_port_type` declared in `MH_Family`, and annotated with all the various MetaH attributes (actually, it is more than just attributes; any piece of information available about the types is captured as an ACME property. For example, we have no use for “`PORT_TYPES`” itself but it is useful to remember the name of the original MetaH enclosure, thus `PORT_TYPES` is saved as a property of both `ANY_TYPE` and `ANOTHER_TYPE`). The resulting ACME port type declarations appear in Figure 2-2 (b)

2.2.2 Subprograms

MetaH subprogram interfaces identify the events that might be raised by a subprogram (out events) and ports that a subprogram sends or receives messages through ([MetaH 98], Section 5.1.3.1.2).

MetaH subprograms are translated just like MetaH processes (Section 2.2.5) except that the ACME component type extends a predefined ACME type “`MH_subprogram`.”

2.2.3 Packages

A package is a collection of executable subprograms. MetaH package interfaces identify the events that might be raised by subprograms in a package; ports that subprograms in a package send or receive messages through; and subprograms and packages that are visible to other objects. ([MetaH 98], Section 5.1.3.1.3).

MetaH packages are translated just like MetaH processes (Section 2.2.5) except that the ACME component type extends a predefined ACME type “`MH_package`.”

2.2.4 Monitors

A monitor is a special type of package object that is shared by multiple processes. ([MetaH 98], Section 5.1.3.1.4).

MetaH monitors are treated like MetaH processes (Section 2.2.5) except that the ACME component type extends a predefined ACME type “`MH_monitor`.”

2.2.5 Processes

MetaH process interfaces identify events that might be raised by a process, ports that a process sends or receives messages through, and subprograms, packages, and monitors that can be shared with other processes ([MetaH 98], Section 5.1.3.2.1)..


```

type package PORT_TYPES is
  ANOTHER_TYPE: type;
  ANY_TYPE: type;
end PORT_TYPES;

type package implementation PORT_TYPES.I80960MC is
attributes
  ANY_TYPE'SourceDataSize := 16 B;
  ANY_TYPE'SourceFile := "port_types.a";
  ANOTHER_TYPE'SourceDataSize := 32 B;
  ANOTHER_TYPE'SourceFile := "port_types.a";
end PORT_TYPES.I80960MC;

```

(a) MetaH Type Package Declaration

```

port type any_type
  extends MH_port_type
  with
    {property MH_Interface_name = "port_types";
     property MH_Implementation_Name = "I80960MC";
     property MH_SourceDataSize = 16;
     property MH_SourceFile = "port_types.a";};

port type another_type
  extends MH_port_type
  with
    {property MH_Interface_name = "port_types";
     property MH_Implementation_Name = "I80960MC";
     property MH_SourceDataSize = 32;
     property MH_SourceFile = "port_types.a";};

```

(b) ACME Port Type Declarations

Figure 2-2: MetaH Type Package and Translation into ACME

MetaH process interfaces are translated into ACME component types extending a predefined ACME type "MH_process." MetaH process implementations are translated into ACME component types extending the ACME type derived from the process interface. For example, the MetaH process interface and implementation in Figure 2-3 (a) are translated into the ACME component types in Figure 2-3 (b).

MH_process includes a pseudo input port "MH_port_event." This "port" is used to establish event-to-port connections.

MetaH event and ports declarations are translated into instances of predefined ACME types "MH_event" or "MH_port." All additional information (e.g., port direction) is translated into ACME properties as illustrated in Figure 2-5.

If a process contains internal components, the implementation structure is translated into an ACME representation, i.e., a System plus (optional) Bindings.

2.2.6 Macros

A MetaH macro allows multiple processes to be grouped to form an abstract object and a macro component can be declared anywhere a process component can be declared ([MetaH 98], Section 5.1.3.2.2).

MetaH macros are treated just like MetaH processes (Section 2.2.5) except that the ACME component type extends a predefined ACME type "MH_macro."

2.2.7 Modes

MetaH mode interfaces identify events that might be raised by a mode, ports that a mode sends or receives messages through, and subprograms, packages, and monitors that can be shared with other modes, macros, and processes. An application can only be executing in one mode of operation at a time ([MetaH 98], Section 5.1.3.2.3).

MetaH modes are treated just like MetaH Macros (Section 2.2.6) except that the ACME component type extends a predefined ACME type "MH_mode."

MetaH modes differ from MetaH macros in that only one mode can be active at a time. There is no counterpart to MetaH's modes in ACME, i.e., there are no dynamic configuration features. We choose instead to treat Modes the same way we treat Macros (i.e., groups of Processes), with an added property "MH_Mode_Class_Name." The semantics of this property is that if multiple "macros" share the same class name, only one may be executing at a time.

```

process P1 is
  p1_input: in port PORT_TYPES.ANY_TYPE;
  update: out port PORT_TYPES.ANOTHER_TYPE;
  feedback: in port PORT_TYPES.ANOTHER_TYPE;
end P1;

process implementation P1.EXAMPLE is
attributes
  self'Period := 25 ms;
  self'SourceFile := "p1_ports.a", "p1.a";
  self'SourceTime := 2 ms;
end P1.EXAMPLE;

```

(a) MetaH Process Declaration

```

component type p1
  extends MH_process
  with
    {port p1_input: MH_port =
      {property MH_port_type = "any_type";
       property MH_port_subclass = MH_in;}};
    port update: MH_port =
      {property MH_port_type = "another_type";
       property MH_port_subclass = MH_out;}};
    port feedback: MH_port =
      {property MH_port_type = "another_type";
       property MH_port_subclass = MH_in;}};
  };

component type p1_example
  extends p1
  with
    {property MH_Period = 25;
     property MH_SourceFiles = {"p1_ports.a", "p1.a"};
     property MH_SourceTime = 2;}};

```

(b) ACME Process Declaration

Figure 2-3: MetaH Process and Translation into ACME

In MetaH, if processes, macros, and modes are components in the same mode, then each process and macro is included in every sibling mode component. The MetaH processes and macros are translated into ACME component types, as usual, but instances of these types get replicated in every sibling mode. This simplifies things because then sibling modes would be self-contained. This is in the spirit of MetaH, where

Modes and macros/processes together form a kind of hierarchical state machine sublanguage. There are submodes and (eventually) a kind of parallel mode composition, the toolset flattens such hierarchical specifications into a final mode transition diagram [Vestal 98].

2.2.8 Applications

The highest level of MetaH specification is an application ([MetaH 98], Section 5.1.3.2.4).

A MetaH application is translated into an ACME component type (a mode, macro, or process) plus an ACME system. The ACME system consists of a single component declaration, an instance of the component type obtained from the MetaH application.

2.2.9 Error Models

An error model declares a set of fault events, a set of error states, and a set of paths that define transitions between error states in response to fault events. Each path declared in an error model is a finite state machine, where the states are error states and the transitions occur in response to fault events. An attribute that can be defined for every source and hardware object is the error model path used to model the response of that object to fault events ([MetaH 98], Section 5.1.3.2.5)).

MetaH error model interfaces are translated into ACME component types extending a predefined ACME type "MH_error_model." MetaH error model implementations are translated into ACME component types extending the ACME type derived from the error model interface.

Error model events and states are translated into instances of predefined ACME types "MH_event" and "MH_error_state," respectively.

Error paths are translated into ACME properties (a sequence of state transitions) as described in Section 2.4.4.2.

2.3 Component Declarations

Each interface and implementation specification can contain zero or more component declarations ([MetaH 98], Section 5.1.4).

Consider the MetaH example in Figure 2-4 (a). This example presents a problem if we handle the interface and the implementation as the usual separate component type declarations and extensions. Not only can we not “extend” an ACME representation, but in addition, in MetaH every object in an interface is also considered to be a component of the implementation. Thus we must wait until we have the MetaH implementation to complete the translation and generate the ACME representation. Note that we must tag the various pieces so we know where they came from in case we want to translate back to MetaH, as shown in Figure 2-4 (b).

2.3.1 Component Classes, Types, and Subclasses

Event, port, and type objects may appear as components of an interface ([MetaH 98], Section 5.1.4.1).

Types are treated as instances of ACME type `MH_port_type`, as shown in Figure 2-2. MetaH event and port declarations are translated into instances of predefined ACME types “`MH_event`” or “`MH_port`” respectively. All additional information (e.g., port direction) is translated into ACME properties as illustrated in Figure 2-5.

2.3.1.1 Port Type and Subclass Declaration

A port must be classified as either an in or out port. Ports are typed and directional, and connections can only be specified between ports whose types and directions are compatible ([MetaH 98], Section 5.1.4.1.1).

The port directions are captured as ACME property “`MH_port_subclass`” with values {“`MH_in`,” “`MH_out`”} defined in `MetaH_Family`.

2.3.1.2 Event Subclass Declaration

MetaH event components (other than those specified in error models) must be subclassified as either in or out events. MetaH event components (other than those specified in error models) may optionally be subclassified as either nudge, signal, interrupt, or mode events ([MetaH 98], Section 5.1.4.1.2).

```

process FOO is
    RESULT: out port PTYPES.INT;
    Q: monitor QUEUE.BOUNDED;
end FOO;

process implementation FOO.BAR is
    COMP: subprogram CONTROLLER.PID;
end FOO.BAR;

```

(a) MetaH Interface and Implementation with Components

```

component type FOO
    extends MH_process
    with
        {port RESULT: MH_port =
            {property MH_port_type = "PTYPES.INT";
             property MH_port_subclass = MH_out;}};
};

component type FOO_BAR extends FOO with
    {representation
        {system MH_some_name =
            component Q =
                new QUEUE_BOUNDED
                extended with
                    (property MH_origin = MH_interface;});
            component COMP =
                new CONTROLLER_PID
                extended with
                    (property MH_origin = MH_implementation;});
        };
    };
};

```

(b) ACME Translation

Figure 2-4: MetaH Interface and Implementation with Components and Translation into ACME

```
m_out: out port PORT_TYPES.ANY_TYPE;
```

(a) MetaH Port Declaration

```
port m_out: MH_port =  
  {property MH_port_type = "any_type";  
   property MH_port_subclass = MH_out;};
```

(b) ACME Port Declaration

Figure 2-5: MetaH Port Declaration and Translation into ACME

The event subclass is captured as the value of ACME property "MH_event_subclass" with values {"MH_nudge," "MH_signal," "MH_interrupt," "MH_mode"} defined in MetaH_Family. The event direction is captured as the value of ACME property "MH_port_subclass" with values {"MH_in," "MH_out"} defined in MetaH_Family.

2.3.1.3 Process Subclass Declaration

Processes are classified as either periodic or aperiodic. A periodic process is dispatched at a fixed frequency specified using the Period attribute. An aperiodic process is dispatched by an event arrival, where the events that can dispatch an aperiodic process are those connected to that process ([MetaH 98], Section 5.1.4.1.3).

The process subclass is captured as the value of ACME property "MH_process_subclass" with values {"MH_periodic," "MH_aperiodic"} defined in MetaH_Family.

The subclassification can appear in either the process implementation or in the individual component. That is, the property could be bound in either place but it must be bound eventually. Event connections to aperiodic processes are described in Section 2.4.2.

2.3.1.4 Mode Subclass Declaration

In any specification that has mode components, exactly one of the modes must be classified as the initial mode, even if there is only one mode component. Event connections can be used to change modes at runtime ([MetaH 98], Section 5.1.4.1.4).

The mode subclass is captured as the value of ACME property “MH_mode_subclass” with values {“MH_initial,” “MH_other”} defined in MetaH_Family. Event connections to mode are described in Section 2.4.2

2.3.1.5 State Subclass Declaration

Exactly one state in an error model must be classified as the initial state, even if there is only one state declared in the error model ([MetaH 98], Section 5.1.4.1.5).

The state subclass is captured as the value of ACME property “MH_state_subclass” with values {“MH_initial,” “MH_other”} defined in MetaH_Family.

2.3.2 Component Visibility and Naming

See the *MetaH Programmer's Manual*, Section 5.1.5 [MetaH 98].

We assume that we are translating a correct MetaH description and that all syntactic and semantic checks have been performed.

2.4 Connection Declarations

Connections in MetaH serve two purposes. They are used to declare connections between the interface elements of the various components in an implementation; they are also used to equivalence (sharing of) objects. For instance, a monitor of one process may be equivalenced to a monitor of another, indicating that the two processes share the same monitor ([MetaH 98], Section 5.1.6).

Connectors are not first class citizens in MetaH. In ACME we make them explicit by declaring them explicitly as instances of connector type “MH_connector.” Port and event connections are mapped to ACME attachments or bindings depending on which components are being connected. In any event, the “citizenship status” of MetaH connectors seems to be mostly a syntactic issue:

Internally, the MetaH tools create a connection object. MetaH connections are not first-class objects as with other ADLs in that the user cannot declare connector implementations with all the power available for declaring, say, process implementations. However, connections are objects in the sense that they can have attributes defined. We expect to add support for a limited form of user-declared connector implementation by allowing a connector to have an associated software component, e.g. a subprogram to do representation or type conversion. The proposed syntax starts to get close to first-class connector objects, e.g. `con_name: port A.In_P <- user_defined (B.Out_P, C.Out_P)`; The exact implementation for "user_defined" is to be inferred from the types of the ports, vaguely like overload resolution. This does not exist yet, it is merely a preliminary proposed extension, but might possibly be of interest to think about [Vestal 98].

2.4.1 Port Connections

Ports may only be connected to other ports ([MetaH 98], Section 5.1.6.1).

For every MetaH port connection between components of an implementation we declare an ACME connector (an instance of component "MH_port_connector," with preassigned roles "MH_source" and "MH_sink") and an ACME attachment section connecting the ports to the roles of the connector, as illustrated in Figure 2-6.

For every MetaH port connection between a component of an implementation and a port of the corresponding interface we declare an ACME binding between the two ports.¹

A MetaH port connection can have an optional identifier. If so, it is saved as an ACME property "MH_identifier" of the ACME connector.

2.4.2 Event Connections

An event may be connected to an aperiodic process, to a mode, or to another event ([MetaH 98], Section 5.1.6.2).

1. Since ACME attachments are declared inside a system and ACME bindings are declared at the same level as the system within a representation, attachments and bindings can not be mixed. Any binding declarations must be postponed until we are done with the attachments and the system declaration, and we are back at the representation level.

```
<<C>> P2.feedback <- P1.update;
```

(a) MetaH Port Connection

```
Connector MH_connector_1 =  
  new MH_port_connector  
  extended with {property MH_identifier = "C"};  
Attachments  
  {p2.feedback to MH_connector_1.MH_sink;  
   p1.update to MH_connector_1.MH_source;};
```

(b) ACME Connector and Attachment Declarations

Figure 2-6: MetaH Port Connection and Translation into ACME

2.4.2.1 Event to Event Connections

Event connections within an object implementation vector events to **out** events and from **in** events declared in that object's interface. Event connections have no attributes and an event connection label serves only for documentation ([MetaH 98], Section 5.1.6.2.1).

Event-to-event connections are treated the same way as port-to-port connections (Section 2.4.1). For every MetaH event-to-event connection between components of an implementation we declare an ACME connector (an instance of component type "MH_event_connector") and an ACME attachment section connecting the ports to the roles of the connector. For every MetaH event-to-event connection between a component of an implementation and an event of the corresponding interface we declare an ACME binding between the two ports.

2.4.2.2 Event to Process Connections

An aperiodic process may be connected to an event. When the event occurs the process will be dispatched ([MetaH 98], Section 5.1.6.2.2).

Every process comes with a pre-declared port, "MH_event_port" (declared in "MH_port"). This allows event-to-process connections to be handled just as port-to-port (or event-to-event) connections (Section 2.4.1).

Multiple events may ultimately dispatch the same aperiodic process, and the same out event raised by a process, monitor, package, or subprogram may ultimately dispatch multiple aperiodic processes. The same out event may ultimately both dispatch one or more aperiodic processes. These are many-to-many connections. Do we need to declare “event” ports that allow unlimited fan-in, fan-out roles?

2.4.2.3 Event to Mode Connections

A mode may be connected to an event. When the event occurs a change to the connected mode will occur ([MetaH 98], Section 5.1.6.2.3).

Every mode comes with a pre-declared port, “MH_event_port” (declared in “MH_mode”). This allows event-to-mode connections to be handled just as port-to-port connections (Section 2.4.1).

2.4.3 Equivalence Connections

Equivalence connections can be used to declare that a pair of monitor component names, package component names, or subprogram component names refer to the same object. An equivalent connection is often used to specify that a subprogram, package, or monitor is shared by multiple processes. ([MetaH 98], Section 5.1.6.3).

Equivalence or sharing of components is not the same as ACME bindings. MetaH Equivalences declare that two component names refer to the same object. ACME bindings declare that two ports are one and the same. We could translate MetaH equivalences into ACME bindings but this would be violating the semantics of the ACME construct. The leading alternative is to not map equivalences into anything and simply add a “MetaH_equivalence” property to the members of an equivalence class so that other tools can tell who they are. The down side is that this might prevent “one-pass” translation since we might have to go back and patch previously processed components with the new property before generating the ACME code.

2.4.4 Path Declarations

Paths define sequencing behaviors of objects ([MetaH 98], Section 5.1.7).

2.4.4.1 Execution Paths

Execution paths may be defined inside process, monitor, package, and subprogram implementation specifications. These are used to describe possible execution control paths through the components of an implementation. Execution paths are typically used in declarations of compute time attributes for processes and their components, and in the computation of stack and heap sizes for a process ([MetaH 98], Section 5.1.7.1).

Execution paths are captured as the value of ACME property “MH_execution_path.” The value of this property is a sequence of component names.

2.4.4.2 Error Paths

Error paths may only be defined inside error model implementation specifications. These are used to describe how the error states of an object may change in response to fault events. An error path takes the form of a list of error state transitions, where each transition identifies a state, an event, and another state. The meaning is that if an object is in the first state, then when the named fault event occurs the object will transition into the second state ([MetaH 98], Section 5.1.7.2).

Error paths are captured as the value of ACME property “MH_error_path” defined in MetaH_Family. The value of this property is a nested sequence of sequences. The inner sequences consist of the three names (state/event/state) in the error state transitions.

2.4.5 Attribute Assignments

The attributes part of an implementation specification contains a sequence of attribute assignments. Each attribute assignment specifies a value for some attribute of a particular object used in an implementation ([MetaH 98], Section 5.1.8).

MetaH attributes are translated into ACME properties. Each attribute name would be tagged with “MH_” to avoid conflicts with predefined names in ACME.

This is the most straightforward translation; alternatively, instead of mapping each attribute to a different ACME property, all MetaH attributes could be captured as the value of a generic ACME property “MH_attribute” with a value of the form <name, value> where the name is the MetaH attribute name.

3 A Complete MetaH Example

```
--
-- Code generated by the ArchEd code generator.
-- Configuration: default
-- Date:          23 September 1994
-- Time:          3:13:33 pm
--
type package PORT_TYPES is
  ANOTHER_TYPE: type;
  ANY_TYPE: type;
end PORT_TYPES;
type package implementation PORT_TYPES.I80960MC is
attributes
  ANY_TYPE'SourceDataSize := 16 B;
  ANY_TYPE'SourceFile := "port_types.a";
  ANOTHER_TYPE'SourceDataSize := 32 B;
  ANOTHER_TYPE'SourceFile := "port_types.a";
end PORT_TYPES.I80960MC;
with type package PORT_TYPES;
macro M is
  m_out: out port PORT_TYPES.ANY_TYPE;
  m_in: in port PORT_TYPES.ANY_TYPE;
end M;
with type package PORT_TYPES;
process P1 is
  p1_input: in port PORT_TYPES.ANY_TYPE;
  update: out port PORT_TYPES.ANOTHER_TYPE;
  feedback: in port PORT_TYPES.ANOTHER_TYPE;
end P1;
process implementation P1.EXAMPLE is
attributes
  self'Period := 25 ms;
  self'SourceFile := "p1_ports.a", "p1.a";
  self'SourceTime := 2 ms;
end P1.EXAMPLE;
with type package PORT_TYPES;
process P2 is
  p2_result: out port PORT_TYPES.ANY_TYPE;
  update: out port PORT_TYPES.ANOTHER_TYPE;
  feedback: in port PORT_TYPES.ANOTHER_TYPE;
end P2;
process implementation P2.EXAMPLE is
```

```

attributes
  self'Period := 50 ms;
  self'SourceFile := "p2_ports.a", "p2.a";
  self'SourceTime := 5 ms;
end P2.EXAMPLE;
macro implementation M.EXAMPLE is
  P2: periodic process P2.EXAMPLE;
  P1: periodic process P1.EXAMPLE;
connections
  P2.feedback <- P1.update;
  P1.feedback <- P2.update;
  m_out <- P2.p2_result;
  P1.p1_input <- m_in;
end M.EXAMPLE;

```

4 The MetaH Example Translated into ACME

```
Family MetaH_Family () =
/* BEGIN STANDARD METAH DECLARATIONS */
..... (see Figure 2-1)
/* BEGIN EXAMPLE SPECIFIC DECLARATIONS */

type package any_type
  extends MH_port_type
  with
    {property MH_Interface_name = "port_types";
     property MH_Implementation_name = "I80960MC";
     property MH_SourceDataSize = 16;
     property MH_SourceFile = "port_types.a";};

type package another_type
  extends MH_port_type
  with
    {property MH_Interface_name = "port_types";
     property MH_Implementation_name = "I80960MC";
     property MH_SourceDataSize = 32;
     property MH_SourceFile = "port_types.a";};

component type M
  extends MH_macro
  with
    {port m_out : MH_port
     = {property MH_port_type = "any_type";
       property MH_port_subclass = MH_out;};
     port m_in : MH_port
     = {property MH_port_type = "any_type";
       property MH_port_subclass = MH_in;};
    };

component type p1
  extends MH_process
  with
    {port p1_input : MH_port
     = {property MH_port_type = "any_type";
       property MH_port_subclass = MH_in;};
     port update : MH_port
```

```

        = {property MH_port_type = "another_type";
          property MH_port_subclass = MH_out;};
port feedback : MH_port
    = {property MH_port_type = "another_type";
      property MH_port_subclass = MH_in;};
};

component type p1_example
    extends p1
    with
        {property MH_Period = 25;
        property MH_SourceFiles = <"p1_ports.a", "p1.a">;
        property MH_SourceTime = 2;};

component type p2
    extends MH_process
    with
        {port p2_result : MH_port
          = {property MH_port_type = "any_type";
            property MH_port_subclass = MH_out;};
        port update : MH_port
          = {property MH_port_type = "another_type";
            property MH_port_subclass = MH_out;};
        port feedback : MH_port
          = {property MH_port_type = "another_type";
            property MH_port_subclass = MH_in;};
        };

component type p2_example
    extends p2
    with
        {property MH_Period = 50;
        property MH_SourceFiles = <"p2_ports.a", "p2.a">;
        property MH_SourceTime = 5;};

Component type M_example
    extends M
    with
        {Representation
        {system MH_little_system =
            {component p2 =
                new p2_example
                extended with
                    {property MH_process_subclass = MH_periodic;};
            component p1 =
                new p1_example
                extended with
                    {property MH_process_subclass = MH_periodic;};
            Connector MH_connector_1 =

```



```

        new MH_connector
        extended with {};
    Attachments
        {p2.feedback to MH_connector_1.MH_sink;
        p1.update to MH_connector_1.MH_source;};
    Connector MH_connector_2 =
        new MH_connector
        extended with {};
    Attachments
        {p1.feedback to MH_connector_2.MH_sink;
        p2.update to MH_connector_2.MH_source;};
}; /* System */
Bindings =
    {m_out to p2.p2_result;
    p1.p1_input to m_in;};
}; /* Representation */
}; /* Type M_Example */
}; /* family */

system MH_system : MetaH_Family =
    {component MH_component = new M_example;};

```

5 Generating Rapide Behavioral Specifications from MetaH Descriptions

MetaH does not include a behavioral specification language as such. A MetaH description implies behavior from the specification of component timing, sequence of operations (execution paths), and state transitions (error paths). If we want to take advantage of available ADLs simulation or verification capabilities, one of the premises of this work and a motivation for ACME, we have essentially three alternatives:

1. Do nothing and translate the MetaH limited behavioral specification into fragments of Rapide specifications. This is not very satisfactory because the behavioral information can be missing or incomplete. A better alternative might be to assign the generation of Rapide specifications to the MetaH timing tool. It performs schedulability analysis and has a better understanding of the behavior of the system.
2. Invent a MetaH behavioral specification language and annotate the relevant components with a new attribute, "behavior," whose value is a string in the new language. This requires writing a translator from this language to Rapide.
3. Adopt the behavioral specification language from an existing ADL and annotate MetaH components with the attribute "behavior" written as a string in that language. The obvious candidate ADL to borrow from is Rapide because it obviates writing a translator. However, if there is already a translator to Rapide from a different ADL (e.g., Wright) we could use it instead.

This aspect of the translation from MetaH to ACME (and then to Rapide) is still incomplete and other alternatives might be considered as the effort progresses.

References

- [ACME 97]** Monroe, Robert; Garlan, David; & While, Dave. *ACME Straw-Manual*, Version 0.1.1 [online]. School of Computer Science, Carnegie Mellon University. Available WWW:
<URL: <http://www.cs.cmu.edu/~acme/>> (November 5, 1997).
- [MetaH 98]** Vestal, Steve. *MetaH Programmer's Manual*, Version 1.22. Honeywell Technology Center. Additional material available WWW:
<URL: http://www.htc.honeywell.com/projects/dssa/dssa_tools.html> (June 1, 1998).
- [Rapide 97]** Rapide Design Team. *Guide to the Rapide 1.0 Language Reference Manuals* [online]. Computer Systems Laboratory, Stanford University. Available WWW:
<URL: <http://poset.Stanford.EDU/rapide/>> (July 17, 1997).
- [Vestal 98]** Vestal, Steve. *Re: MetaH to ACME translation* [email to M.R. Barbacci]. Available email: mrb@sei.cmu.edu, April 6, 1998.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (leave blank)		2. REPORT DATE July 1998	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Mapping MetaH into ACME		5. FUNDING NUMBERS C — F19628-95-C-0003	
6. AUTHOR(S) Mario R. Barbacci and Charles B. Weinstock			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-98-SR-006	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.b DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) This report explores the translation of MetaH into ACME as a first step into the translation of MetaH to other architecture description languages (e.g., Rapide) to take advantage of any toolsets developed for the target language. We start by comparing the meta-models of ACME and MetaH, we establish mapping rules for each MetaH construct, and we present a full MetaH example taken from the MetaH Library at Honeywell. The report concludes with a brief description of possible alternative paths to obtain (limited) Rapide behavioral specifications from MetaH timing and sequencing of operations.			
14. SUBJECT TERMS architecture description language, behavioral specifications, components, connectors, structural specifications		15. NUMBER OF PAGES 28	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL